МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій

Кафедра комп'ютерної інженерії та інформаційних систем

ЛАБОРАТОРНА РОБОТА №2

з дисципліни «Теорія і технології проектування спеціалізованих операційних систем»

ЛРКІ. 180102.22.01.02 ПЗ

Галузь знань \_\_\_\_\_\_\_\_\_\_12 – Інформаційні технології\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Спеціальність \_\_\_\_\_\_\_\_\_123 –Комп’ютерна інженерія\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Виконав:

студент 1 курсу, групи КІ2м-22-1 Біньковський Я.В.

(Підпис)

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лисенко С.М.

(Підпис)

Хмельницький – 2023 рік

**Тема:**

Проєктування спеціалізованої RTOS для Інтернету речей на платформі Raspberry Pi. Реалізація базових функцій ядра. Додавання до проєкту драйверів GPIO та UART та перенесення його на цільову платформу. Реалізація функції завантаження ядра по UART. Додавання функцій апаратного таймера для усунення циклічних затримок в драйвері GPIO. Тестування на реальній платформі

**Завдання:**

Реалізувати етапи 5-7 проєктування спеціалізованої RTOS для платформи Raspberry Pi3. Протестувати поточну версію ядра та зробити відповідні висновки. Пояснити відмінності реалізації для платформи Raspberry Pi4. Оформити звіт з виконаної роботи.

**Виконання завдання:**

Етап 5.

На цьому етапі розробляється інфраструктура для додавання перших справжніх драйверів пристроїв. Після включення безпечних глобальних змінних ми зможемо використовувати реальну UART замість консолі QEMU. Для побудови для RPi 4 потрібно додати BSP = rpi4 до кожної цілі. Додамо ознаку driver::interface::DeviceDriver для абстрагування BSP реалізацій драйвера від коду ядра. Драйвери зберігаються в src/bsp/device\_driver, і їх можна повторно використовувати між BSP.

Було введено драйвер GPIO, який пінмуксує (тобто направляє сигнали від внутрішніх SoC до фактичних пінів HW) UART PL011 для RPi. Оскільки апаратне забезпечення RPi 3 і RPi 4 відрізняється, то драйвер має враховувати це в програмному забезпеченні. Драйвер PL011Uart реалізує властивості console::interface::\* і використовується як основний вихід системної консолі.

Код реалізації:

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

DEV\_SERIAL ?= /dev/ttyUSB0

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

EXEC\_TEST\_DISPATCH = ruby ../common/tests/dispatch.rb

EXEC\_MINITERM = ruby ../common/serial/miniterm.rb

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

DOCKER\_ARG\_DIR\_COMMON = -v $(shell pwd)/../common:/work/common

DOCKER\_ARG\_DEV = --privileged -v /dev:/dev

# DOCKER\_IMAGE defined in include file (see top of this file).

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

DOCKER\_TEST = $(DOCKER\_CMD) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

ifeq ($(shell uname -s),Linux)

DOCKER\_CMD\_DEV = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_ARG\_DEV)

DOCKER\_MINITERM = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

endif

.PHONY: all doc qemu miniterm clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

miniterm:

@$(DOCKER\_MINITERM) $(EXEC\_MINITERM) $(DEV\_SERIAL)

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

--section .rodata \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

.PHONY: test test\_boot

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

test\_boot test:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

test\_boot: $(KERNEL\_BIN)

$(call color\_header, "Boot test - $(BSP)")

@$(DOCKER\_TEST) $(EXEC\_TEST\_DISPATCH) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

test: test\_boot

endif

Файл driver.rs

use crate::{

println,

synchronization::{interface::Mutex, NullLock},

};

const NUM\_DRIVERS: usize = 5;

struct DriverManagerInner {

next\_index: usize,

descriptors: [Option<DeviceDriverDescriptor>; NUM\_DRIVERS],

}

pub mod interface {

pub trait DeviceDriver {

fn compatible(&self) -> &'static str;

unsafe fn init(&self) -> Result<(), &'static str> {

Ok(())

}

}

}

pub type DeviceDriverPostInitCallback = unsafe fn() -> Result<(), &'static str>;

#[derive(Copy, Clone)]

pub struct DeviceDriverDescriptor {

device\_driver: &'static (dyn interface::DeviceDriver + Sync),

post\_init\_callback: Option<DeviceDriverPostInitCallback>,

}

pub struct DriverManager {

inner: NullLock<DriverManagerInner>,

}

static DRIVER\_MANAGER: DriverManager = DriverManager::new();

impl DriverManagerInner {

/// Create an instance.

pub const fn new() -> Self {

Self {

next\_index: 0,

descriptors: [None; NUM\_DRIVERS],

}

}

}

impl DeviceDriverDescriptor {

pub fn new(

device\_driver: &'static (dyn interface::DeviceDriver + Sync),

post\_init\_callback: Option<DeviceDriverPostInitCallback>,

) -> Self {

Self {

device\_driver,

post\_init\_callback,

}

}

}

pub fn driver\_manager() -> &'static DriverManager {

&DRIVER\_MANAGER

}

impl DriverManager {

pub const fn new() -> Self {

Self {

inner: NullLock::new(DriverManagerInner::new()),

}

}

pub fn register\_driver(&self, descriptor: DeviceDriverDescriptor) {

self.inner.lock(|inner| {

inner.descriptors[inner.next\_index] = Some(descriptor);

inner.next\_index += 1;

})

}

fn for\_each\_descriptor<'a>(&'a self, f: impl FnMut(&'a DeviceDriverDescriptor)) {

self.inner.lock(|inner| {

inner

.descriptors

.iter()

.filter\_map(|x| x.as\_ref())

.for\_each(f)

})

}

pub unsafe fn init\_drivers(&self) {

self.for\_each\_descriptor(|descriptor| {

if let Err(x) = descriptor.device\_driver.init() {

panic!(

"Error initializing driver: {}: {}",

descriptor.device\_driver.compatible(),

x

);

}

if let Some(callback) = &descriptor.post\_init\_callback {

if let Err(x) = callback() {

panic!(

"Error during driver post-init callback: {}: {}",

descriptor.device\_driver.compatible(),

x

);

}

}

});

}

pub fn enumerate(&self) {

let mut i: usize = 1;

self.for\_each\_descriptor(|descriptor| {

println!(" {}. {}", i, descriptor.device\_driver.compatible());

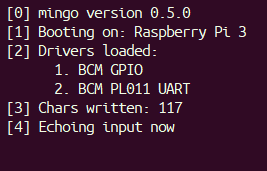
i += 1;

});

}

}

Результат:



Етап 6.

На даному етапі розробники створюють ланцюговий завантажувач, що дозволить зручно завантажувати ядро через UART. Раніше для завантаження нового двійкового файлу на пристрій використовувалася SD-карта, але це було незручно, оскільки потрібно було записувати на неї кожен новий файл. Ланцюговий завантажувач - це програма, яка запускається на пристрої при його включенні і дозволяє завантажувати нове ядро через UART.

Код реалізації:

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

DEV\_SERIAL ?= /dev/ttyUSB0

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

CHAINBOOT\_DEMO\_PAYLOAD = demo\_payload\_rpi3.img

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

CHAINBOOT\_DEMO\_PAYLOAD = demo\_payload\_rpi4.img

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

# This parses cargo's dep-info file.

# https://doc.rust-lang.org/cargo/guide/build-cache.html#dep-info-files

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

EXEC\_TEST\_MINIPUSH = ruby tests/chainboot\_test.rb

EXEC\_MINIPUSH = ruby ../common/serial/minipush.rb

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

DOCKER\_ARG\_DIR\_COMMON = -v $(shell pwd)/../common:/work/common

DOCKER\_ARG\_DEV = --privileged -v /dev:/dev

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

DOCKER\_TEST = $(DOCKER\_CMD) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

ifeq ($(shell uname -s),Linux)

DOCKER\_CMD\_DEV = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_ARG\_DEV)

DOCKER\_CHAINBOOT = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

endif

.PHONY: all doc qemu chainboot clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu qemuasm:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

qemuasm: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU with ASM output")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN) -d in\_asm

endif

chainboot: $(KERNEL\_BIN)

@$(DOCKER\_CHAINBOOT) $(EXEC\_MINIPUSH) $(DEV\_SERIAL) $(CHAINBOOT\_DEMO\_PAYLOAD)

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

--section .rodata \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

.PHONY: test test\_boot

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

test\_boot test:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

test\_boot: $(KERNEL\_BIN)

$(call color\_header, "Boot test - $(BSP)")

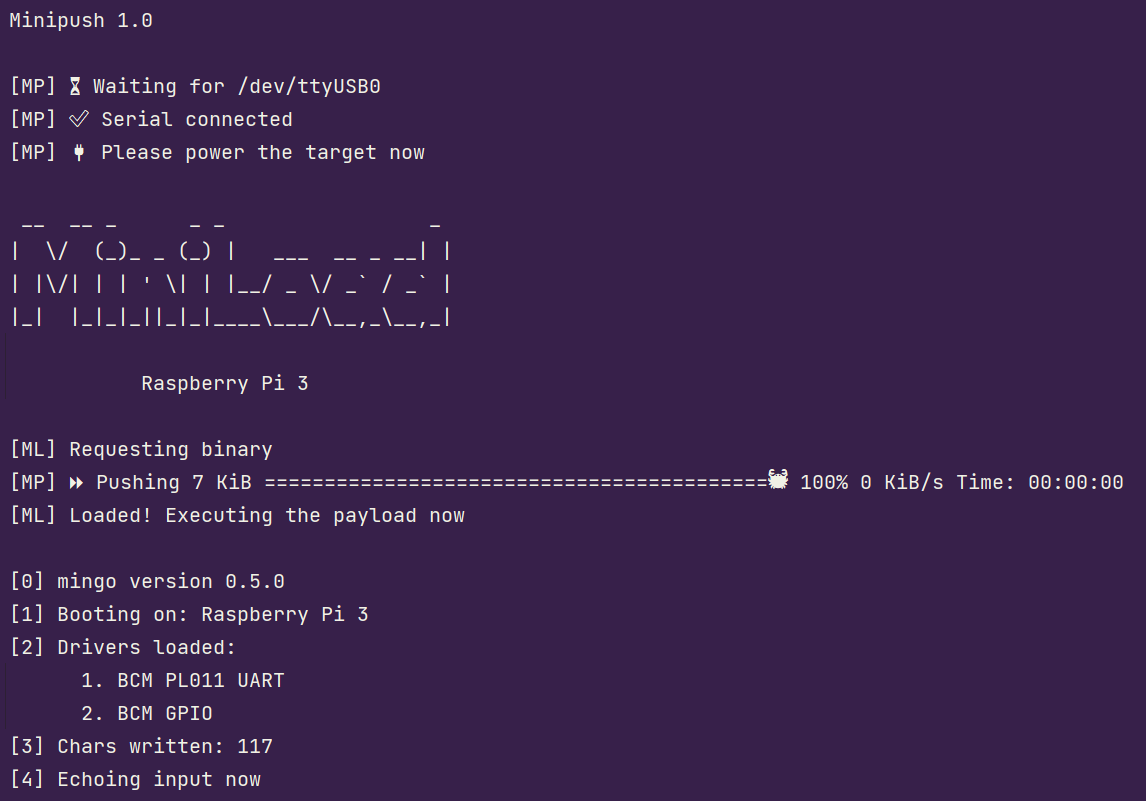
@$(DOCKER\_TEST) $(EXEC\_TEST\_MINIPUSH) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) \

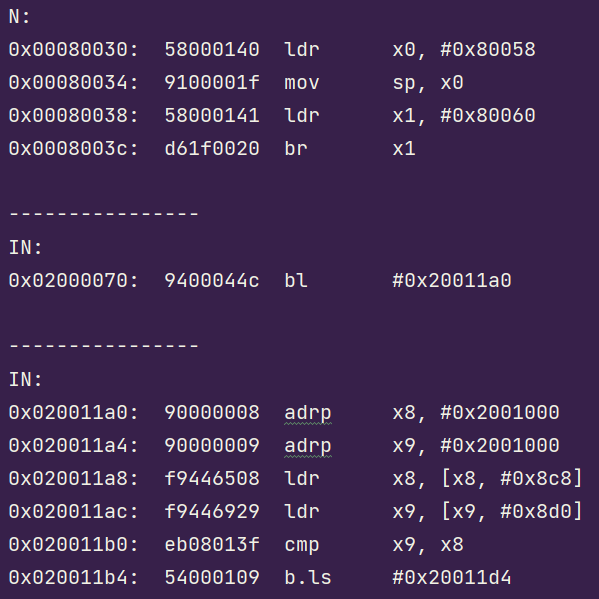
-kernel $(KERNEL\_BIN) $(CHAINBOOT\_DEMO\_PAYLOAD)

test: test\_boot

endif

Результат:





Етап 7.

На цьому етапі додаються абстракції для апаратного таймера та реалізують їх для архітектурного таймера ARM у \_arch/aarch64. Це дозволяє створювати нові функції таймера, які використовуються для анотації друку по UART мітками часу та для усунення циклічних затримок в драйвері пристрою GPIO. Використання міток часу дозволяє відстежувати точний час виконання операцій, що може бути корисним для налагодження системи та виявлення проблем. Додавання макросу warn!() дозволяє виводити повідомлення про помилки, що допомагає виявляти та виправляти проблеми в системі. В цілому, на цьому етапі розробники працюють над покращенням точності та функціональності системи.

Код реалізації:

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

DEV\_SERIAL ?= /dev/ttyUSB0

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

# This parses cargo's dep-info file.

# https://doc.rust-lang.org/cargo/guide/build-cache.html#dep-info-files

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

EXEC\_TEST\_DISPATCH = ruby ../common/tests/dispatch.rb

EXEC\_MINIPUSH = ruby ../common/serial/minipush.rb

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

DOCKER\_ARG\_DIR\_COMMON = -v $(shell pwd)/../common:/work/common

DOCKER\_ARG\_DEV = --privileged -v /dev:/dev

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

DOCKER\_TEST = $(DOCKER\_CMD) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

ifeq ($(shell uname -s),Linux)

DOCKER\_CMD\_DEV = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_ARG\_DEV)

DOCKER\_CHAINBOOT = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

endif

.PHONY: all doc qemu chainboot clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

chainboot: $(KERNEL\_BIN)

@$(DOCKER\_CHAINBOOT) $(EXEC\_MINIPUSH) $(DEV\_SERIAL) $(KERNEL\_BIN)

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

--section .rodata \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

.PHONY: test test\_boot

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

test\_boot test:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

test\_boot: $(KERNEL\_BIN)

$(call color\_header, "Boot test - $(BSP)")

@$(DOCKER\_TEST) $(EXEC\_TEST\_DISPATCH) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

test: test\_boot

endif

Файл time.rs

mod arch\_time;

use core::time::Duration;

pub struct TimeManager;

static TIME\_MANAGER: TimeManager = TimeManager::new();

pub fn time\_manager() -> &'static TimeManager {

&TIME\_MANAGER

}

impl TimeManager {

pub const fn new() -> Self {

Self

}

pub fn resolution(&self) -> Duration {

arch\_time::resolution()

}

pub fn uptime(&self) -> Duration {

arch\_time::uptime()

}

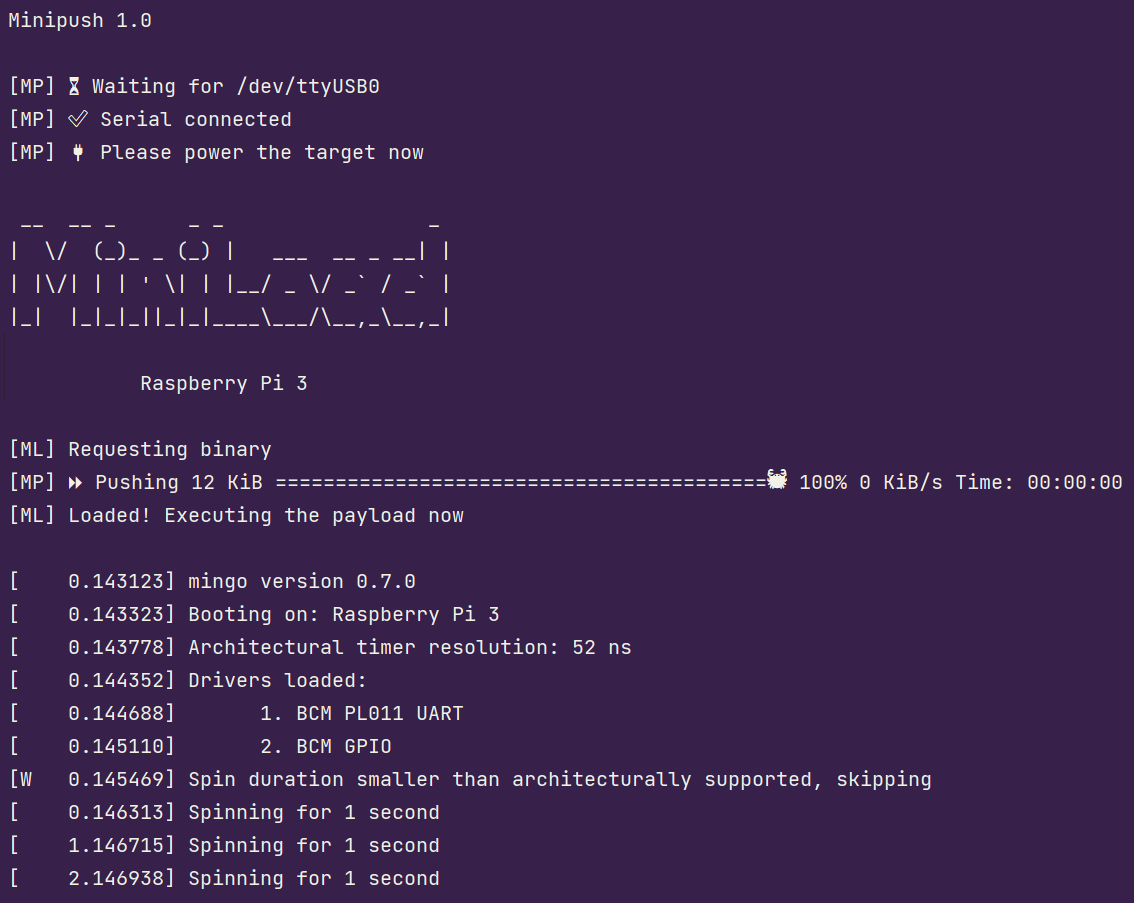
pub fn spin\_for(&self, duration: Duration) {

arch\_time::spin\_for(duration)

}

}

Результат виконаної програми:



**Висновок:**

На останніх трьох кроках було додано ряд важливих функцій і абстракцій для забезпечення правильної роботи ядра. Зокрема, був реалізований ланцюговий завантажувач, що дозволяє зручно завантажувати ядро через UART без необхідності записувати кожен новий двійковий файл на SD-карту. Були також додані абстракції для апаратного таймера, які підвищують точність драйвера пристрою GPIO і дозволяють анотувати друк по UART мітками часу. Останні кроки дозволили покращити функціональність ядра та забезпечити більш точну роботу пристроїв.

На Raspberry Pi3 та Raspberry Pi4 архітектурний таймер ARM є однаковим, тому основний процес додавання абстракцій для таймера та їхня реалізація повинна бути схожою на обох платформах. Однак, можуть відрізнятись деталі реалізації для взаємодії з конкретними версіями процесора та іншими апаратними компонентами.

Крім того, Raspberry Pi4 має більш потужний процесор та більше оперативної пам'яті, що може позитивно позначитись на продуктивності і швидкодії доданого коду.